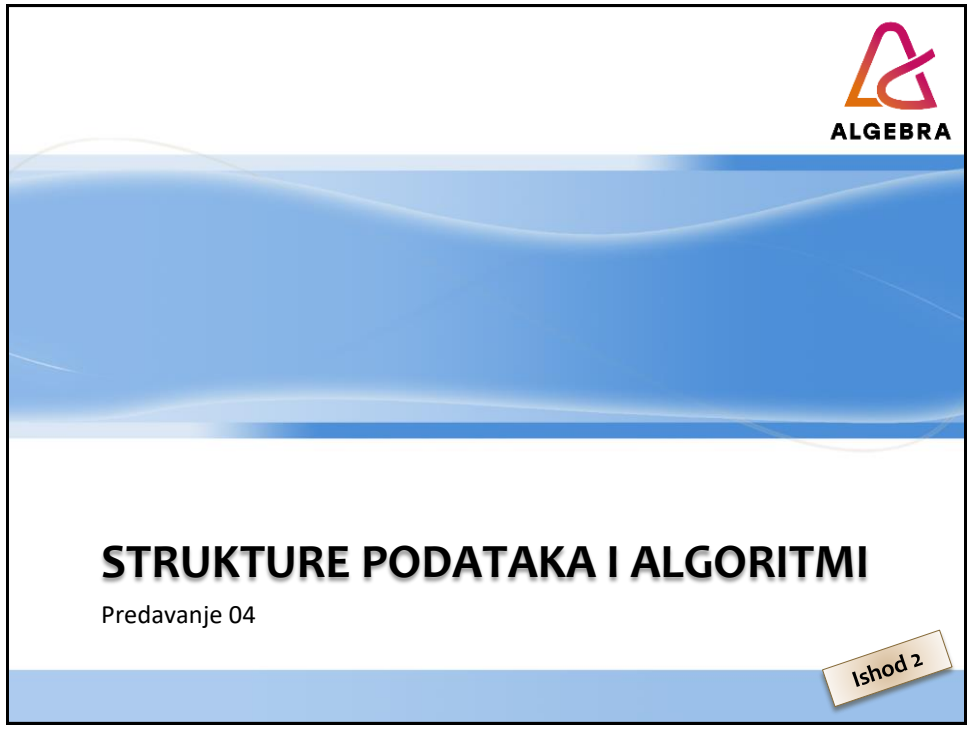


1

Apstraktni i konkretni tipovi podataka

- Apstraktni tip podataka (engl. ADT, *abstract data type*) predstavlja korisnikovu želju za funkcionalnostima
  - Definira tip podataka u smislu podržanih operacija i njihove složenosti
  - Ne kaže ništa o načinu kako će biti implementiran
  - Opisuje tip podataka sa stajališta korisnika tipa podataka
- Konkretni tip podataka (engl. *concrete data type*) predstavlja implementaciju apstraktnog tipa
  - Primjerice: `vector<T>` je implementacija liste u C++
    - U C# je to `List<T>`, u Javi je to `ArrayList`, u Pythonu je to `List`, u JavaScriptu je to `Array`, u Rustu je to `Vec`, itd.

ALGEBRA

2

LISTA

Operacije na listi

- ADT lista nigdje nije formalno definirana
  - „The ADT List is a linear sequence of an arbitrary number of items” (izvor: doc.ic.ac.uk)
- Popis mogućih operacija na listi:
  - Izrada prazne liste
  - Umetanje novog elementa na neku poziciju u listi
  - Uklanjanje elementa s neke pozicije u listi
  - Provjera je li lista prazna ili ne
  - Dohvaćanje elementa na nekoj poziciji u listi
  - Dohvaćanje broja elemenata u listi

Ostale moguće operacije

- Osmislite još tri operacije koje imaju smisla na listi
  - Umetanje elementa na kraj
  - Dohvaćanje prvog elementa
  - Dohvaćanje zadnjeg elementa
  - Uklanjanje svih elemenata iz liste
  - Traženje prvog pojavljivanja zadane vrijednosti u listi
  - Traženje svih pojavljivanja zadane vrijednosti u listi
  - Dohvaćanje sljedećeg elementa iza neke pozicije
  - Dohvaćanje prethodnog elementa iza neke pozicije
  - …

ALGEBRA

5

VEKTOR

ALGEBRA

6

Vektor kao konkretna ADT lista

- Vektor je C++ implementacija ADT lista
  - Sadrži niz metoda koje predstavljaju operacije nad listom, primjerice:
    - Možemo napraviti praznu listu (konstruktor)
    - Možemo umetnuti novi element na neku poziciju u listi (metoda `insert`)
    - Možemo ukloniti element s neke pozicije u listi (metoda `remove`)
    - ...
- Za razliku od ADT liste, vektor je konkretan i spreman za korištenje
  - Izveden kao generička klasa `vector<T>`

Princip rada vektora

- Vektor je objekt koji sadrži pokazivač na dinamičko polje na hrpi
  - Elementi vektora su poslagani jedan iza drugoga u memoriji
  - Možemo im pristupati pokazivačima kao kod običnih polja
- Dinamičko polje na hrpi može „rasti” prema potrebi

Princip rada vektora

- „Rast" dinamičkog polja na hrpi se odvija automatski, kad treba dodati novi element, a mjesta više nema
  - Treba umetnuti novi element, a cijelo dinamičko polje je puno
  - Događa se sljedeća (skupa) operacija:
    - Alocira se novo, veće dinamičko polje i u njega se stavlja novi element
    - Svi elementi iz starog polja se kopiraju/pomiču u novo polje
    - Staro polje se otpušta
  - Optimizacija: rast vektora se dešava eksponencijalno
    - Često je nova veličina za 50% veća od stare
    - „*Libraries can implement different strategies for growth to balance between memory usage and reallocations*"
    - Cilj: izbjeći rast pri svakom umetanju i osigurati amortizirano *O*(1) za umetanje na kraj

ALGEBRA

9

Izrada i uništavanje vektora (1/2)

- Postoji šest osnovnih načina izrade vektora:
  - `vector<int> jedan;`
    - Kreira prazni vektor (*default*)
  - `vector<int> dva(n);`
    - Kreira vektor od *n* elemenata inicijaliziranih na *defaultnu* vrijednost (*fill*)
  - `vector<int> tri(n, val);`
    - Kreira vektor od *n* elemenata, svaki je kopija od *val* (*fill*)
  - `vector<int> cetiri(tri.begin(), tri.end());`
    - Kreira vektor kopiranjem elemenata iz zadanog raspona (*range*)
      - Prva vrijednost je početna adresa (uzima se i element na toj adresi)
      - Druga vrijednost je zadnja adresa (element na toj adresi se ne uzima)

ALGEBRA

10

Izrada i uništavanje vektora (2/2)

- `vector<int> pet(tri);`
  - Kreira vektor na način da kopira sve elemente iz zadanog vektora (*copy*)
- `vector<int> sest({ 11, 22, 33 });`
  - Kreira vektor na način da kopira sve elemente iz inicijalizacijske liste (*initializer list*)

- Vektor se automatski uništava završetkom funkcije u kojoj je deklariran
  - Ako vektor čuva objekte, na svakom se poziva destruktor

Primjer

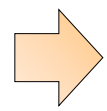
- Objasnite ponašanje:

```
class Bla {
public:
    ~Bla() {
        cout << "Destruktor" << endl;
    }
};

int main() {

    vector<Bla> sest;
    sest.push_back(Bla());
    sest.push_back(Bla());
    sest.push_back(Bla());

    return 0;
}
```



```
Destruktor
Destruktor
Destruktor
Destruktor
Destruktor
Destruktor
Destruktor
Destruktor
Destruktor
Destruktor
```


Kopiranje vektora

- `operator=` kopira sadržaj desnog vektora u lijevi
  - Prethodni sadržaj lijevog vektora se uništava (prepisivanjem ili uništavanjem)
  - Veličina lijevog vektora se može promijeniti nakon kopiranja
  - Da bi kopiranje bilo moguće, oba vektora moraju biti istog tipa T
- Primjer:

```
vector<int> jedan(3, 404);
vector<int> dva(5, 701);

dva = jedan;

for (unsigned i = 0; i < dva.size(); i++) {
     cout << dva[i] << endl;
}
```


ALGEBRA

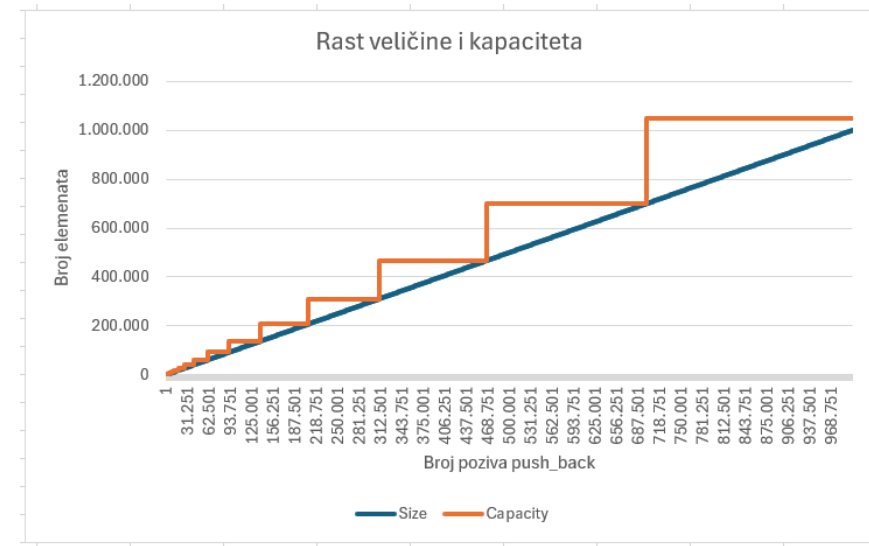
Veličina i kapacitet vektora

- Kod vektora v razlikujemo dvije mjere:
  - `v.size()` vraća broj elemenata stavljenih u vektor od strane korisnika
  - `v.capacity()` vraća veličinu alociranog dinamičkog polja (također izraženu u broju elemenata)
  - Kad `size()` treba preteći `capacity()`, vektor raste
- Primjer:

```
vector<int> jedan;
for (unsigned i = 0; i < 100; i++) {
    cout << "size=" << jedan.size() << " (capacity=" <<
        jedan.capacity() << ")" << endl;
    jedan.push_back(i);
}
```


ALGEBRA

Rast veličine i kapaciteta



ALGEBRA

15

Usporedba performansi

- Koja je razlika u performansama:

```
vector<int> v1;
for (unsigned i = 0; i < n; i++) {
    v1.push_back(i+1);
}
```

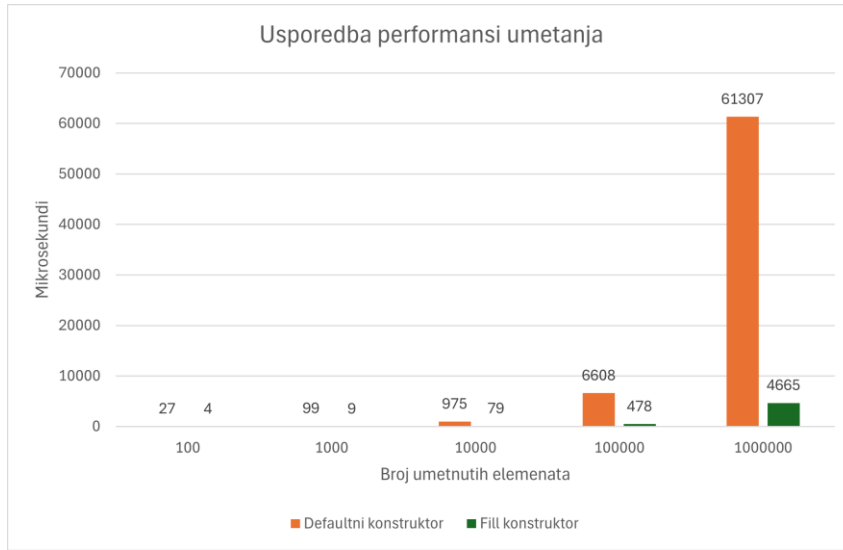
- i

```
vector<int> v2(n);
for (unsigned i = 0; i < n; i++) {
    v2[i] = i + 1;
}
```


ALGEBRA

16

Usporedba performansi



Ručna promjena veličine i kapaciteta vektora

- Veličinu i kapacitet možemo i eksplicitno mijenjati:
  - `v.resize(n);`
    - Mijenja veličinu vektora na točno *n* elemenata
    - Ako je *n* manji od trenutne veličine, odbacuju se elementi s kraja
      - Kapacitet se ne mijenja
    - Ako je *n* veći od trenutne veličine, dodaju se elementi na kraj
      - Ako je *n* veći i od trenutnog kapaciteta, vektor raste
  - `v.reserve(n);`
    - Mijenja kapacitet vektora tako da može sadržavati barem *n* elemenata
      - Ako je *n* veći od trenutnog kapaciteta, vektor raste
      - Ako je *n* manji od trenutnog kapaciteta, metoda ne radi ništa

Primjer

```
vector<int> v;
v.push_back(10);
v.push_back(20);
v.push_back(30);

v.resize(0);
cout << "s=" << v.size() << ", c=" << v.capacity() << endl;

v.resize(38);
cout << "s=" << v.size() << ", c=" << v.capacity() << endl;

v.reserve(100);
cout << "s=" << v.size() << ", c=" << v.capacity() << endl;

v.reserve(75);
cout << "s=" << v.size() << ", c=" << v.capacity() << endl;
```


ALGEBRA

19

Pristup elementima

- Vektor nudi nekoliko načina pristupa elementima:
  - `v[i]` vraća referencu na element na mjestu *i*
    - Ako je *i* izvan opsega, ponašanje nije definirano 
  - `v.at(i)` vraća referencu na element na mjestu *i*
    - Ako je *i* izvan opsega, baca iznimku tipa `out_of_range` 
  - `v.front()` vraća referencu na prvi element
    - Ako je vektor prazan, ponašanje nije definirano 
  - `v.back()` vraća referencu na zadnji element
    - Ako je vektor prazan, ponašanje nije definirano 

ALGEBRA

20

Primjer

```
vector<int> jedan(5);

for (unsigned i = 0; i < jedan.size(); i++) {
      jedan[i] = (i + 1) * 10;
}

for (unsigned i = 0; i < jedan.size(); i++) {
      cout << jedan.at(i) << " ";
}
cout << endl;

cout << jedan.front() << " " << jedan.back() << endl;
```


Modifikatori vektora (1/3)

- Modifikatori vektora mijenjaju količinu elemenata:
  - `v.assign()` je sličan konstruktorima *fill*, *range* i *initializer list*

```
v.assign(7, 100);
v.assign(x.begin(), x.end());
v.assign({ 11, 22, 33 });
```

    - Svi elementi prethodno sadržani u vektoru se uništavaju
    - Ako će nova veličina biti veća od trenutnog kapaciteta, vektor će rasti
  - `v.push_back(val)`
    - Dodaje kopiju od *val* na kraj vektora
    - Može uzrokovati rast vektora
    - Preferirani način punjenja vektora jer ne zahtijeva pomicanje preostalih elemenata

Modifikatori vektora (2/3)

- `v.pop_back()`
  - Uklanja i uništava zadnji element (i smanjuje veličinu za 1)
  - Preferirani način uklanjanja iz vektora
- `v.insert()` umeće jedan ili više elemenata na zadanu poziciju:
  - Prvi parametar je pozicija, ostali su slični konstruktorima:

```
v.insert(v.begin() + 3, 99);
v.insert(v.begin() + 3, 10, 99);
v.insert(v.begin() + 3, v.begin(), v.end());
v.insert(v.begin() + 3, { 11, 22, 33 });
```

  - Može uzrokovati rast vektora
  - Ako se umetanje ne radi na kraj vektora, svi ostali elementi iza novih će biti pomaknuti u desno – loše po performanse

Modifikatori vektora (3/3)

- `v.erase()` uklanja jedan ili više elemenata:

```
v.erase(v.begin() + 3);
v.erase(v.begin() + 3, v.end());
```

  - Uklanja i uništava element na zadanoj poziciji ili u zadanom rasponu (i smanjuje veličinu za broj uklonjenih elemenata)
  - Vraća iterator na sljedeći element iza obrisanih
    - Svi postojeći iteratori koji pokazuju na obrisana mjesta i mjesta iza njih postaju neispravni
  - Ako se uklanjanje ne radi s kraja vektora, svi ostali elementi iza uklonjenih će biti pomaknuti u lijevo – loše po performanse
- `v.clear()` kompletno prazni vektor i uništava sve elemente
  - Veličina odlazi na 0, kapacitet se može, ali i ne mora promijeniti (ovisi o implementaciji)
  - Ako se radi o objektima, na svakom elementu se poziva destruktor

Primjer

```
vector<int> v(5, 0);

v.pop_back();
v.pop_back();

v.push_back(10);

v.insert(v.begin(), 2, 20);

v.erase(v.begin() + 2);
v.erase(v.begin() + 2);

for (unsigned i = 0; i < v.size(); i++) {
	cout << v[i] << " ";
}
cout << endl;
```


Ostale važnije metode

- `v.empty()` vraća je li vektor prazan ili ne

ITERATORI, BRISANJE I KONSTRUIRANJE OBJEKTA NA MJESTU

Iteratori

- Iterator je standardni način za pristup podacima koji se nalaze u nekom kontejneru (vektoru, mapi, listi, …)
- Iterator `it` je svaki objekt koji ima sljedeće karakteristike:
  - Omogućuje pristup nekom elementu (`*it`)
  - Omogućuje odlazak na sljedeći element (`++it`)
  - Opcionalno, omogućuje odlazak na prethodni element (`--it`)
- Pokazivač je također iterator jer zadovoljava gornje uvjete
- Neki kontejneri imaju i alternativne načine pristupa elementima, ali su iteratori univerzalni za sve kontejnere
  - Primjerice, `[]` ili `at()`

Iteratori vektora

- Vektor sadrži nekoliko vrsta iteratora, a najkorisniji su:
  - `vector<T>::iterator` je klasa čiji ++ vodi prema kraju
  - `vector<T>::reverse_iterator` je klasa čiji ++ vodi prema početku (reverzni iterator)
- Metode koje vraćaju iteratore:
  - `v.begin()` – vraća iterator na prvi element
  - `v.end()` – vraća iterator na prvi element iza kraja
  - `v.rbegin()` – vraća iterator na reverzni početak (a to je zadnji element)
  - `v.rend()` – vraća iterator na reverzni kraj (a to je element točno ispred prvog elementa u vektoru)

Primjer

```
vector<int> v;

for (int i = 10; i <= 50; i += 10) {
    v.push_back(i);
}

for (vector<int>::iterator it = v.begin(); it != v.end(); ++it) {
    cout << *it << endl;
}

for (vector<int>::reverse_iterator it = v.rbegin();
                                        it != v.rend(); ++it) {
    cout << *it << endl;
}
```


Brisanje iz vektora (1/2)

- Kako bismo iz vektora v izbrisali sve parne brojeve?

```
vector<int> v({ 11, 22, 33, 44, 55 });
```

- Kod brisanja iz vektora moramo voditi računa o sljedećem:
  - „erase … invalidates iterators and references at or after the point of the erase ...”
  - Zbog toga ovaj način brisanja nije ispravan:

```
for (auto it = v.begin(); it != v.end(); ++it) {
    if (*it % 2 == 0) {
        v.erase(it);
    }
}
```

  - Razlog: nakon prvog brisanja iterator `it` više nije ispravan i ne smijemo ga povećati s ++

ALGEBRA

Brisanje iz vektora (2/2)

- Dva glavna načina za brisanje:
  - Mala modifikacija neispravnog brisanja kako bi postalo ispravno:

```
for (auto it = v.begin(); it != v.end(); ) {
    if (*it % 2 == 0) {
        it = v.erase(it);
    }
    else {
        ++it;
    }
}
```

- Ugrađeni algoritam `remove_if`

ALGEBRA

Ima li razlike između ++it i it++?

- Sljedeći primjer demonstrira rad prefiks i postfiks operatora (i tu nema ničega novoga za nas):

```
int x = 10;
cout << x++ << endl;
cout << ++x << endl;
```

- Postoji li razlika između sljedeće dvije petlje?

```
for (auto it = v.begin(); it != v.end(); it++) {
    cout << *it << endl;
}

for (auto it = v.begin(); it != v.end(); ++it) {
    cout << *it << endl;
}
```

- Razlika u rezultatu ne postoji – obje ispisuju isto
  - Međutim, razlike u performansama mogu postojati

Koja je razlika između ++it i it++?

- Prefiks na objektu radi ovako:
  - Promijeni originalni objekt
  - Vrati referencu na originalni objekt
- Postfiks na objektu radi ovako:
  - Kreira novi privremeni objekt kopiranjem originala
  - Ažurira originalni objekt
  - Vrati kopiju
- Postfiks koristi dodatno kopiranje, što je obično loše
  - Na ugrađenim tipovima podataka nema razlike
  - Postoji šansa da će optimizator izbjeći kopiranje
  - Ako koristimo prefiks, ne možemo pogriješiti!

Nepotrebno kopiranje objekata

- Neka je zadana struktura:

```
struct Pravokutnik {
      Pravokutnik(int s, int v) {
            this->sirina = s;
            this->visina = v;
      }
      int sirina;
      int visina;
};
```

- Kako možemo dodati novi pravokutnik na kraj vektora?

```
vector<Pravokutnik> vp;
Pravokutnik p(17, 4);
vp.push_back(p);
```

- Koliko smo objekata kreirali i možemo li problem riješiti pametnije?

Metode `emplace()` i `emplace_back()`

- Metoda `emplace()` se ponaša jednako kao i `insert()`, osim što ne kopira već konstruira objekt na ciljnom mjestu
  - Metoda `emplace_back()` se ponaša jednako, samo što objekt dodaje na kraj vektora (tj. ne prima poziciju kao parametar)
- Obje metode primaju varijabilni broj parametara:
  - Prvi parametar je uvijek pozicija (samo za `emplace()`)
  - Ostali parametri su u stvari vrijednosti koje idu u odgovarajući konstruktor
- Rješenje našeg problema s prošlog slajda:

```
vector<Pravokutnik> vp;
vp.emplace_back(17, 4);
```


Složenost nekih operacija

Metoda	Složenost
vector<T> v;	O(1)
vector<T> v(n);	O(n)
vector<T> v(n, value);	O(n)
vector<T> v(begin, end);	O(n)
v[i];	O(1)
v.at(i);	O(1)
v.size();	O(1)
v.empty();	O(1)
v.begin();	O(1)
v.end();	O(1)
v.front();	O(1)
v.back();	O(1)
v.capacity();	O(1)

Metoda	Složenost
v.push_back(value);	O(1)
v.insert(iterator, value);	O(n)
v.pop_back();	O(1)
v.erase(iterator);	O(n)
v.erase(begin, end);	O(n)

Zadatak

1. Implementirajte svoj jednostavni vektor cijelih brojeva. Neka strategija povećanja kapaciteta bude da je novi kapacitet uvijek za 50% veći od dosadašnjeg. Na vektoru definirajte sljedeće operacije:
   a) Kreiranje vektora i njegova inicijalizacija pomoću liste
   b) Dohvat veličine i kapaciteta
   c) Umetanje elementa na kraj
   d) Dohvat elementa na mjestu *i*

Source.cpp

```
MojVektor mv({ 11, 22, 33, 44, 55 });
mv.push_back(66);
mv.push_back(77);
mv.push_back(88);
mv.push_back(99);

cout << "s=" << mv.size() << ", c=" << mv.capacity() << endl;
for (int i = 0; i < mv.size(); ++i) {
    cout << mv.at(i) << endl;
}
```


MojVektor.h

```
#pragma once
#include <initializer_list>

class MojVektor {
private:
    int* brojevi;
    int s;
    int c;
    void grow();

public:
    MojVektor(std::initializer_list<int> il);
    ~MojVektor();
    int size();
    int capacity();
    void push_back(int value);
    int at(int i);
};
```


MojVektor.cpp (1/3)

```
#include "MojVektor.h"

MojVektor::MojVektor(std::initializer_list<int> il) {
    brojevi = new int[il.size()];
    int i = 0;
    for (auto it = il.begin(); it != il.end(); ++it) {
        brojevi[i++] = *it;
    }
    s = il.size();
    c = il.size();
}

MojVektor::~MojVektor() {
    delete[] brojevi;
}
```


MojVektor.cpp (2/3)

```
int MojVektor::size() {
    return s;
}

int MojVektor::capacity() {
    return c;
}

void MojVektor::push_back(int value) {
    if (c == s) {
        grow();
    }
    brojevi[s++] = value;
}

int MojVektor::at(int i) {
    return brojevi[i];
}
```


MojVektor.cpp (3/3)

```
void MojVektor::grow() {
    // Alociraj novo polje
    c = c * 1.5;
    int* novi = new int[c];

    // Prepiši vrijednosti staro => novo
    for (int i = 0; i < s; i++) {
        novi[i] = brojevi[i];
    }

    // Otpusti staro.
    delete[] brojevi;

    // Prekopiraj adresu novog polja
    brojevi = novi;
}
```


Dodatni materijali

- Dodatni materijali su dostupni na:
  - Vectors
    - https://youtu.be/rBTX1m6a9vg
  - Iterators, removals and constructing in-place
    - https://youtu.be/pow_7Aod9MU
  - Implementing your own vector
    - https://youtu.be/8Zpfg-uVtyE